

# Characterizing and computing stable models of logic programs: the non-stratified case

**G. Brignoli**

Università di Milano  
Dip. di Scienze dell'Informazione  
I-20135 Milano, Italy  
brigno@ntboss.tesi.dsi.unimi.it

**S. Costantini**

Università di L'Aquila  
Dip. di Matematica Pura e Applicata  
I-67100 L'Aquila, Italy  
stefcost@univaq.it

**O. D'Antona and A. Proveti**

Università di Milano  
Dip. di Scienze dell'Informazione  
I-20135 Milano, Italy  
{dantona, provetti}@dsi.unimi.it

Stable Logic Programming (SLP) is an emergent, alternative style of logic programming: each solution to a problem is represented by a stable model of a deductive database/function-free logic program encoding the problem itself. Several implementations now exist for stable logic programming, and their performance is rapidly improving. To make SLP generally applicable, it should be possible to check for consistency (i.e., existence of stable models) of the input program before attempting to answer queries. In the literature, only rather strong sufficient conditions have been proposed for consistency, e.g., stratification. This paper extends these results in several directions. First, the syntactic features of programs, viz. cyclic negative dependencies, affecting the existence of stable models are characterized, and their relevance is discussed. Next, a new graph representation of logic programs, the Extended Dependency Graph (EDG), is introduced, which conveys enough information for reasoning about stable models (while the traditional Dependency Graph does not). Finally, we show that the problem of the existence of stable models can be reformulated in terms of coloring of the EDG.

## Background definitions

The stable model semantics (Gelfond & Lifschitz 1988) is a view of logic programs as sets of inference rules (more precisely, default inference rules), where a stable model is a set of atoms closed under the program itself. Alternatively, one can see a program as a set of constraints on the solution of a problem, where each stable model represents a solution compatible with the constraints expressed by the program. Consider the simple program  $\{q \leftarrow \text{not } p, \text{not } c. \ p \leftarrow \text{not } q. \ p \leftarrow c.\}$ . For instance, the first rule is read as “assuming that both  $p$  and  $c$  are false, we can conclude that  $q$  is true.” This program has two stable models. In the first,  $q$  is true while  $p$  and  $c$  are false; in the second,  $p$  is true while  $q$  and  $c$  are false.

Unlike with other semantics, a program may have no stable model, i.e., be contradictory, like the following:  $\{a \leftarrow \text{not } b. \ b \leftarrow \text{not } c. \ c \leftarrow \text{not } a.\}$ , where no set of atoms is closed under the rules. It is important to make sure that a program admits stable models before attempting to perform deduction. Inconsistency may arise, realistically, when programs are combined: if they share atoms, a subprogram like that above may surface in the resulting program.

In this paper we consider, essentially, the language  $DATALOG^-$  for deductive databases, which is more restricted than traditional logic programming. As discussed in (Marek & Truszczyński 1999), this restriction is not a limitation at this stage.

A rule (clause)  $\rho$  is defined as usual, and can be seen as composed of a conclusion  $head(\rho)$ , and a set of conditions  $body(\rho)$ , the latter divided into positive conditions  $pos(\rho)$  and negative conditions  $neg(\rho)$ .

For syntax and semantics of logic programs with negation (general, or normal logic programs), and for the definition of Dependency Graph (DG), the reader may refer for instance to (Apt & Bol 1994) and to the references therein.

For the sake of clarity however, let us report the definition of stable models. We start from the subclass of positive programs, i.e. those where, for every rule  $\rho$ ,  $neg(\rho) = \emptyset$ .

### Definition 1 (Stable model of positive programs)

The stable model  $a(\Pi)$  of a positive program  $\Pi$  is the smallest subset of  $\mathbb{B}_\Pi$  such that for any rule  $a \leftarrow a_1, \dots, a_m$  in  $\Pi$ :  $a_1, \dots, a_m \in a(\Pi) \Rightarrow a \in a(\Pi)$ .

Positive programs are unambiguous, in that they have a unique stable model, which coincides with that obtained applying other semantics.

### Definition 2 (Stable models of programs)

Let  $\Pi$  be a logic program. For any set  $S$  of atoms, let  $\Pi^S$  be a program obtained from  $\Pi$  by deleting (i) each rule that has a formula ‘not  $A$ ’ in its body with  $A \in S$ , and (ii) all formulae of the form ‘not  $A$ ’ in the bodies of the remaining rules.

$\Pi^S$  does not contain “not,” so that its stable model is already defined. If this stable model coincides with  $S$ , then we say that  $S$  is a stable model of  $\Pi$ . In other words, the stable models of  $\Pi$  are characterized by the equation:  $S = a(\Pi^S)$ .

Programs which have a unique stable model are called categorical.

In the literature, the main (sufficient) condition to ensure the existence of stable models is call-consistency, which is summarized as follows: no atom depends on itself via an odd number of negative conditions.

**Proposition 1** (Dung 1992) A [normal] logic program has a stable model if it is call-consistent.

More results along the lines of Proposition 1 are found in (Dung 1992). However, this condition is quite restrictive, as there are programs with *odd cycles* (in the sense described above) that have one or more stable models. See Example 1 below.

For the sake of simplicity, in this paper we consider *kernel programs*, that are general logic programs where: (i) there are no positive conditions, i.e. for every clause  $\rho$ ,  $\text{pos}(\rho) = \emptyset$ ; (ii) every atom which is the head of a rule must also appear in the body of some rule (possibly the same one).

From any program  $\Pi$ , a kernel program  $\text{ker}(\Pi)$  can be obtained, which is equivalent to  $\Pi$  as far as characterizing stable models.

## The relationship between cycles and stable models

As discussed above, we are interested in programs that are not stratified (*unstratified* programs), and do not satisfy call-consistency. We will speak of an *even* (resp. *odd*) cycle referring to a even (resp. odd) number of rules organized like  $\{a \leftarrow \text{not } b, b \leftarrow \text{not } a.\}$  (resp.  $\{c \leftarrow \text{not } e, e \leftarrow \text{not } f, f \leftarrow \text{not } c.\}$ ). This Section is devoted to the analysis of the type and number of cycles appearing in a program, and their *connections*, i.e. roughly, rules involving atoms that appear in different cycles, which we call *handles*. We argue that the form of cycles and connections is *the key factor* affecting the existence –and the number– of stable models. In fact, the dependency graph makes neither the cycles, nor the connections explicit.

**Example 1** Consider the following programs,  $\Pi_1$ ,  $\Pi_2$  and  $\Pi_3$ :

$$\begin{array}{lll} p \leftarrow \text{not } p, \text{not } e. & p \leftarrow \text{not } p. & p \leftarrow \text{not } p, \text{not } e. \\ a \leftarrow \text{not } b. & p \leftarrow \text{not } e. & a \leftarrow \text{not } b. \\ b \leftarrow \text{not } a. & a \leftarrow \text{not } b. & b \leftarrow \text{not } a. \\ e \leftarrow \text{not } f. & b \leftarrow \text{not } a. & e \leftarrow \text{not } f. \\ f \leftarrow \text{not } h. & e \leftarrow \text{not } f. & f \leftarrow \text{not } h. \\ h \leftarrow \text{not } e. & f \leftarrow \text{not } h. & h \leftarrow \text{not } e, \text{not } a. \\ h \leftarrow \text{not } a. & h \leftarrow \text{not } e, \text{not } a. & \end{array}$$

It is easy to see that the dependency graphs of the three programs in Example 1 coincide. However,  $\Pi_1$  has the stable model  $\{b, h, e\}$  while instead  $\Pi_2$  has the stable model  $\{a, f, p\}$  and  $\Pi_3$  has no stable models at all. Why do they have such a diverse semantics? The reason relies in the different decomposition of the three programs into cycles. The programs above are divided into cycles as follows, where *OC* and *EC* denote odd and even cycle, respectively, and literals appearing either in square brackets or in braces correspond to different kinds of handles. Consider the following partitions of  $\Pi_1$  and  $\Pi_2$ , respectively:

$$\begin{array}{ll} OC_1 : \{ p \leftarrow \text{not } p, [\text{not } e]. & OC_1 : \{ p \leftarrow \text{not } p. \\ EC_1 : \{ a \leftarrow \text{not } b. & H_2 : \{ p \leftarrow \{\text{not } e.\} \\ & b \leftarrow \text{not } a. \\ OC_2 : \{ e \leftarrow \text{not } f. & EC_1 : \{ a \leftarrow \text{not } b. \\ & f \leftarrow \text{not } h. & b \leftarrow \text{not } a. \\ & h \leftarrow \text{not } e. \\ H_1 : \{ h \leftarrow \{\text{not } a.\}. & OC_2 : \{ e \leftarrow \text{not } f. \\ & f \leftarrow \text{not } h. \\ & h \leftarrow \text{not } e, [\text{not } a]. \end{array}$$

The literals in braces are called *OR handles* of the cycle. Consider program  $\Pi_1$ . Literal *not a* in  $H_1$  is an OR handle for  $OC_2$ . Now, consider a putative stable model  $S$ ; if  $a \notin S$ , we can say that “handle  $H_1$  is true.” Then, atom  $g$  is forced to be in  $S$  and, consequently,  $OC_2$  has, w.r.t.  $S$ , the stable model  $\{g, e\}$ . Literal *not e*, instead, is an *AND handle* (indicated in square brackets) of the odd cycle  $OC_1$ : if it is false (i.e.,  $e \in S$ ), it forces  $p$  to be false, and  $OC_1$  “has the empty model,” and  $p \notin S$ .

Similar considerations can be made on  $\Pi_2$ , even though it has a different structure: literal *not a* in this case is an AND handle to  $OC_2$  (while in  $\Pi_1$  it is an OR handle, instead); if *not a* is true then the odd cycle  $OC_2$  is contradictory, and determines the inconsistency of the whole program. If, on the other hand, *not a* is false, then  $g$  is forced to be false, and consequently  $OC_2$  has the stable model  $\{f\}$ . This means moreover that the OR handle *not e* of  $OC_1$  is true, and thus  $p$  is true: therefore the contradiction  $p \leftarrow \text{not } p$ , which could determine the inconsistency of the whole program, is made harmless. Finally, the reader can easily check that program  $\Pi_3$  has the odd cycle  $OC_2$  unconstrained (no handles); thus,  $\Pi_3$  has no stable models. A formal assessment of cycles will be part of the forthcoming extended version of this paper.

At this point, it is however important to notice that one rule may belong to several cycles at once.

**Example 2** Let  $\Pi_4$  :

$$\begin{array}{ll} p \leftarrow \text{not } p, \text{not } q. & a \leftarrow \text{not } b. \\ q \leftarrow \text{not } q, \text{not } p. & b \leftarrow \text{not } a. \\ q \leftarrow \text{not } v. & z \leftarrow \text{not } z, \text{not } k. \\ v \leftarrow \text{not } w. & k \leftarrow \text{not } l. \\ w \leftarrow \text{not } a. & l \leftarrow \text{not } k. \end{array}$$

In  $\Pi_4$ , the following cycles are found:

$$C_1 = \{p \leftarrow \text{not } p, \text{not } q.\}$$

This is an odd cycle ( $p$  depends on itself).

$$C_2 = \{q \leftarrow \text{not } q, \text{not } p.\}$$

This is an odd cycle ( $q$  depends on itself). Moreover, the former two rules together form also an even cycle, where  $p$  depends on  $q$  and vice versa, i.e.:

$$C_3 = \{q \leftarrow \text{not } p, \text{not } q. p \leftarrow \text{not } q, \text{not } p.\}$$

Now,

$$C_4 = \{a \leftarrow \text{not } b. b \leftarrow \text{not } a.\}$$

is an even cycle, while

$$C_5 = \{z \leftarrow \text{not } z, \text{not } k.\}$$

is an odd cycle ( $z$  depends on itself). Finally,

$$C_6 = \{k \leftarrow \text{not } l. l \leftarrow \text{not } k.\}$$

is an even cycle,  $k$  depends on  $l$  and vice versa. There are clauses, namely  $q \leftarrow \text{not } v$ ,  $v \leftarrow \text{not } w$ , and  $w \leftarrow \text{not } a$ , which do not belong to any cycle. Notice however that they

can be seen as forming a chain connecting cycles. In fact, since the first atom in the chain is  $q$ , which belongs to cycles  $C_2$  and  $C_3$ , in a way this chain forms two bridges: one between  $C_2$  and  $C_4$ , the other between  $C_3$  and  $C_4$ .

In Example 2 above, clause  $q \leftarrow \text{not } v$  is called an *auxiliary rule* of cycles  $C_2$  and  $C_3$ , since its conclusion  $q$  is an atom belonging to these cycles. As mentioned above, auxiliary rules can belong to a *bridge* connecting different cycles. For the sake of simplicity, we can assume that all bridges have unitary length, i.e. that all bridges reduce to an auxiliary rule. In fact, what is important is which cycle is connected to which, while the intermediate steps of the chain do not affect the existence and number of stable models.

In the rest of the paper, we will say that a cycle  $C$  is *constrained* if it has an handle. Then, a cycle with no handle is called *unconstrained*.

### From cycles to stable models

In order to reason about the existence of the stable models of  $\Pi$ , it is useful to reason about the existence of the stable models of its composing cycles.

**Definition 3** An extended cycle  $EC$  is a set of rules composed of one cycle  $C$  together with all its auxiliary clauses.

**Proposition 2** A program  $\Pi$  has a unique decomposition into extended cycles  $\{EC_1, \dots, EC_r\}$ ,  $r \geq 1$ .

**Definition 4** Let  $C$  be an extended cycle, and let  $H_C$  be the set of all the atoms occurring in some of the handles of  $C$ . Let  $\mathcal{I} \in 2^{H_C}$ . A completed extended cycle  $CC$  is a set of rules composed of one extended cycle  $C$ , where atoms in  $\mathcal{I}$  are added as unit clauses.

Notice that adding to  $EC$  some of the atoms of  $H_C$  (which are atoms occurring in the handles of  $C$ ) corresponds to making an hypothesis about truth/falsity of the handles of  $C$ . For any extended cycle  $EC$ , there are  $2^{H_C}$  corresponding completed cycles, each one corresponding to a different hypothesis on the handles. Correspondingly, there are several ways of decomposing  $\Pi$  into completed cycles  $\{CC_1, \dots, CC_r\}$ ,  $r \geq 1$ . What we intend to show is the direct relation between the stable models of the completed extended cycles and the stable models of the overall program. Indeed, a completed cycle, taken as a program *per se*, may or may not have stable models.

**Theorem 1** A program  $\Pi$  with decomposition into cycles  $\{C_1, \dots, C_r\}$  has stable models only if there exists a set of completed extended cycles  $\{CC_1, \dots, CC_r\}$  of  $\Pi$  such that every  $CC_i$ ,  $i \leq r$ , has a stable model.

For any decomposition of  $\Pi$  into completed extended cycles, we are interested only in those sets  $\{S_1, \dots, S_r\}$  of stable models of, respectively,  $\{CC_1, \dots, CC_r\}$  which agree on shared atoms. In other words, a consistent set of partial stable models contains one stable model for each of the extended cycles of the decomposition, and there are no  $S_i, S_j$  assigning opposite truth values to some atom.

**Theorem 2** An interpretation  $I$  of  $\Pi$  is a stable model if and only if  $I = \{S_1 \cup \dots \cup S_r\}$  where  $\{S_1, \dots, S_r\}$  is a consistent set of stable models for a decomposition  $\{CC_1, \dots, CC_r\}$  of  $\Pi$  into completed extended cycles.

Then, from the stable models of the composing cycles, we are able to obtain the stable models of the program. Correspondingly, if we study the conditions for the existence of stable models of the (extended) cycles, we can find conditions for the existence of stable models of  $\Pi$ .

It is easy to see that whenever a cycle  $C_\alpha$  is constrained, then there exists a corresponding completed, extended cycle  $CC_\alpha$  which is a locally stratified program; thus,  $CC_\alpha$  has a unique stable model, which also coincide with the Well-founded model.

Assume instead that  $\Pi$  contains an unconstrained cycle  $C$ . In this case, the unique completed extended cycle associated to  $C$  is  $C$  itself (*trivial* completed extended cycle). If  $C$  is even, then it has the two stable models:

$$M_C^1 = \{a_i : i \leq n, i = 2k + 1\}$$

$$M_C^2 = \{a_j : j \leq n, j = 2k\}$$

Vice versa, if  $C$  is odd there are no stable models. In conclusion, we can state the following propositions.

**Proposition 3** An unconstrained even cycle always has a corresponding (trivial) completed extended cycle with stable models.

**Proposition 4** An unconstrained odd cycle has no corresponding completed extended cycles with a stable model.

These considerations allow us to formulate some useful necessary and sufficient conditions for the existence of stable models.

In our framework, for instance, it becomes easy to reformulate the result in (Dung 1992) saying that every call-consistent program has stable models. Moreover, it is also easy to establish the following.

**Proposition 5** A program  $\Pi$  has a stable model only if every odd composing cycle  $C$  is constrained.

There are situations however, where the odd cycles are constrained, but *still no stable model exists*. This happens whenever all possible decompositions of  $\Pi$  lead to sets of partial stable models which are not consistent. I.e., there are cycles which require opposite truth values of some atom, in order to have stable models, e.g.

$$p \leftarrow \text{not } p, \text{not } a.$$

$$q \leftarrow \text{not } q.$$

$$q \leftarrow \text{not } a.$$

It is possible to identify sufficient conditions for the existence of stable models, based on ruling out these situations *constructively*. This is discussed below as well as in our forthcoming work.

### A new graph representation

In order to reason more directly and more efficiently about cycles and handles, we introduce a new graph representation of programs, since the usual DG is not adequate to this aim. On this graph, we should be able of: detecting by means of efficient algorithms the syntactic features of programs w.r.t. the classification sketched above; reasoning about the existence and the number of stable models; computing them.

This new graph is similar to the DG, except it is more accurate for negative dependencies, and thus has been called EDG (Extended Dependency Graph).

The definition is based upon distinguishing among rules defining the same atom, i.e., having the same head. To establish this distinction, we assign to each head an upper index, starting from 0, e.g.,  $\{a \leftarrow c, \text{not } b. a \leftarrow \text{not } d.\}$  becomes  $\{a^0 \leftarrow c^0, \text{not } b^0. a^1 \leftarrow \text{not } d^0.\}$ . However, for the sake of clarity, we write  $a_i$  instead of  $a_i^{(0)}$ . The main idea underlying the next definition is to create, for any atom  $a$ , as many vertices in the graph as the rules with head  $a$  (labeled  $a, a^1, a^2$  etc.).

**Definition 5** (Extended dependency graph) (EDG)

For a logic program  $\Pi$ , its associated Extended Dependency Graph  $EDG(\Pi)$  is the directed finite labeled graph  $\langle V, E, \{+, -\} \rangle$  defined below. The main idea underlying the definition of EDG is that of creating, for any atom  $a$ , as many vertices in the graph as the rules with head  $a$  (labeled  $a, a^1, a^2$  etc.).

- V.1** For each rule in  $\Pi$  there is a vertex  $a_i^{(k)}$ , where  $a_i$  is the name of the head and  $k$  is the index of the rule in the definition of  $a_i$ .
- V.2:** for each atom  $u$  never appearing in a head, there is a vertex simply labeled  $u$ ;
- E.1:** for each  $c_j^{(l)} \in V$ , there is a positive edge  $\langle c_j^{(l)}, a_i^{(k)}, + \rangle$ , if and only if  $c_j$  appears as a positive condition in the  $k$ -th rule defining  $a_i$ , and
- E.2:** for each  $c_j^{(l)} \in V$ , there is a negative edge  $\langle c_j^{(l)}, a_i^{(k)}, - \rangle$ , if and only if  $c_j$  appears as a negative condition in the  $k$ -th rule defining  $a_i$ .

The definition of EDG extends that of DG in the sense that for programs where atoms are defined by at most one rule the two coincide. Consider in Figure 1 the EDGs of the programs in Example 1. As all conditions in  $\Pi_1, \Pi_2$  and  $\Pi_3$  are negative, for the sake of simplicity, the ‘-’ labels are omitted from edges.

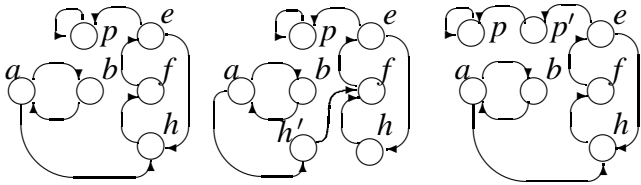


Figure 1:  $EDG(\Pi_3)$  (left),  $EDG(\Pi_1)$  (center) and  $EDG(\Pi_2)$  (right).

Notice that both  $DG(\Pi_1), DG(\Pi_2)$  and  $DG(\Pi_3)$  correspond to  $EDG(\Pi_3)$ .

The main idea underlying the definition of EDG is that of creating, for any atom  $a$ , as many vertices in the graph as the rules with head  $a$  (labeled  $a, a^1, a^2$  etc.). For instance, in  $EDG(\Pi_1)$  (center of Figure 1) arc  $\langle h, f, - \rangle$  represents rule  $\{f \leftarrow \text{not } h.\}$ . On the graph, we clearly see the cycles, and

also the handles. In fact, rule  $\{f \leftarrow \text{not } h.\}$  must be represented by the two arcs  $\langle h, f, - \rangle$  and  $\langle h', f, - \rangle$  since truth of  $h$  may depend on any of its defining rules; the second one is auxiliary to the cycle, and corresponds to an OR handle. Therefore, the cycle has an OR handle *if and only if there is an incoming arc originated in a duplication of one of the atoms of the cycle*. In this case, the arc  $\langle h', f, - \rangle$  represents the OR handle of  $OC_2$ . In the same graph, arc  $\langle e, p, - \rangle$  represents instead the AND handle of  $OC_1$ . Therefore, a cycle has an AND handle *if and only if there exists an incoming arc into that cycle in the EDG, originated in (any duplication of) an atom not belonging to the cycle itself*. A cycle with no incoming arcs is unconstrained.

It is easy to see that the EDG of a program is isomorphic to the program itself. Consequently, the EDG conveys enough information for reasoning about stable models of the program.

### Coloring EDGs

This section describes how the EDG can be used to study the stable models in terms of graph coloring. Let us define a *coloring* as an assignment of nodes of a graph to colors, e.g.  $\nu : V \rightarrow \{\text{green}, \text{red}\}$ . An interpretation corresponds to a coloring, where all the true atoms are green, and all the others are red.

We now specify which colorings we intend to rule out, since they trivially correspond to inconsistencies.

**Definition 6** (non-admissible coloring)

A coloring  $\nu : V \rightarrow \{\text{green}, \text{red}\}$  is non-admissible for  $\langle V, E \rangle = EDG(\Pi)$  if and only if

1.  $\exists i. \nu(v_i) = \text{green}$  and  $\exists j. (v_i, v_j, -) \in E$  and  $\nu(v_j) = \text{green}$ , or
2.  $\exists i. \nu(v_i) = \text{red}$  and  $\forall j. (v_j, v_i, -) \in E$  and  $\nu(v_j) = \text{red}$ .

To sum it up, **green nodes cannot be adjacent and edges to a red node cannot all come from red nodes**.

A coloring for  $EDG(\Pi)$  is admissible unless it is not admissible. A partial coloring is admissible if all its *completions* (intuitively) are.

**Example 3** What are the admissible colorings for  $EDG(\Pi_1)$  in Example 1?

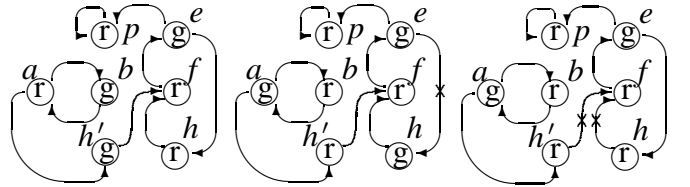


Figure 2: An admissible coloring of  $EDG(\Pi_1)$  (on the left) and two not admissible ones (center and right, resp.) of  $EDG(\Pi_1)$  with g=green, r=red and X=admissibility violation.

In the center coloring above, arc  $\langle e, h, - \rangle$  violates admissibility. In fact, it corresponds to rule  $h \leftarrow \text{not } e$  in  $\Pi_1$ .

If  $e$  is true/green, then by the rule  $h$  cannot be concluded true/green. As a matter of fact, both  $e$  and  $h$  are true in the stable model of  $\Pi_1$ , but the truth of  $h$  comes from, intuitively, labeling  $h'$  green in the first coloring. In the right coloring above, admissibility is violated by arcs  $\langle h', f, - \rangle$  and  $\langle h, f, - \rangle$  which, together, represent the rule  $f \leftarrow \text{not } h$  of  $\Pi_1$ . When all  $h$ s are red, we conclude  $h$  false and –by the above rule–,  $f$  true/green.

Now, we are able to define a notion of admissible coloring for EDG's of Kernel programs.

**Theorem 3** *An interpretation  $\mathcal{I}$  is a stable model of  $\Pi$  if and only if it corresponds to an admissible coloring of  $EDG(\Pi)$ .*

We are implementing a practical system that computes stable models on the EDG (Brignoli et al. 1999). The coloring procedure is, experimentally, very sensitive to the choice of heuristic methods for starting the coloring itself from “relevant” nodes. In fact, presently the choice of the starting nodes is guided by the concept of extended cycle described earlier: we try identify nodes corresponding to crucial handles, and start from them. A main topic for our research now is clearly the study of new heuristic methods, as well as adapting existing solutions from graph theory.

### Acknowledgments

Thanks to Chitta Baral and Michael Gelfond for constant encouragement in the pursuit of this research.

### References

- Apt, K. R. and Bol, R., 1994. *Logic programming and negation: a survey*, J. of Logic Programming, 19/20.
- Baral, C. and Gelfond, M., 1994. *Logic programming and knowledge representation*, J. of Logic Programming, 19/20.
- Brignoli G., Costantini S. and Proveti A., 1999. *A Graph Coloring algorithm for stable models generation*. Univ. of Milan Technical Report, submitted for publication.
- Costantini S., 1995. *Contributions to the stable model semantics of logic programs with negation*, Theoretical Computer Science, 149.
- Cholewiński P., Marek W. and Truszczyński M., 1996. *Default reasoning system DeReS*. Proc. of KR96, Morgan-Kaufman, pp. 518-528.
- Cholewiński P. and Truszczyński M., 1996. *Extremal problems in logic programming and stable model computation*. Proc. of IJCSLP'96, pp. 408-422. Also in J. of Logic Programming, 38(1999), pp. 219-242.
- Dimopoulos Y., 1996. *On Computing Logic Programs*, J. of Automated Reasoning, 17:259-289.
- Dung P.M., 1992. *On the Relation between Stable and Well-Founded Semantics of Logic Programs*, Theoretical Computer Science, 105.
- Dung P.M. and Kanchanasut, 1989. *Logic programming and stable model computation*, Proc. of NACLP'89.
- Eiter, T., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F., 1997. *A deductive system for non-monotonic reasoning*. Proc. Of the 4 th LPNMR Conference, Springer Verlag, LNCS 1265, pp. 363-374.
- Gelfond, M. and Lifschitz, V., 1988. *The stable model semantics for logic programming*, Proc. of 5th ILPS conference, pp. 1070-1080.
- Marek, W., and Truszczyński M., 1991. *Autoepistemic Logic*. The Journal of the ACM, 38:588-619.
- Marek, W., and Truszczyński M., 1999. *Stable models and an alternative logic programming paradigm*. The Journal of Logic Programming.
- Niemelä I. and Simons P., 1998. *Logic programs with stable model semantics as a constraint programming paradigm*. Proc. of NM'98 workshop. Extended version submitted for publication.
- Saccà D. and Zaniolo C., 1997. *Deterministic and Non-Deterministic Stable Models*. J. of Logic and Computation.
- Simons P., 1997. *Towards Constraint Satisfaction through Logic Programs and the Stable Models Semantics*, Helsinki Univ. of Technology R.R. A:47.
- Subrahmanian, V.S., Nau D., and Vago C., 1995. *WFS + branch and bound = stable models*, IEEE Trans. on Knowledge and Data Engineering, 7(3):362-377.
- Van Gelder A., Ross K.A. and Schlipf J., 1990. *The Well-Founded Semantics for General Logic Programs*. Journal of the ACM Vol. 38 N. 3.
- Chen W., and Warren D.S., 1996. *Computation of stable models and its integration with logical query processing*, IEEE Trans. on Data and Knowledge Engineering, 8(5):742-747.